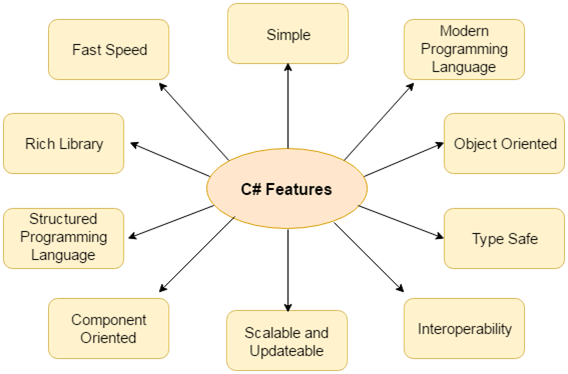
# C# Features

C# is object oriented programming language. It provides a lot of **features** that are given below.

1. Simple
2. Modern programming language
3. Object oriented
4. Type safe
5. Interoperability
6. Scalable and Updateable
7. Component oriented
8. Structured programming language
9. Rich Library
10. Fast speed



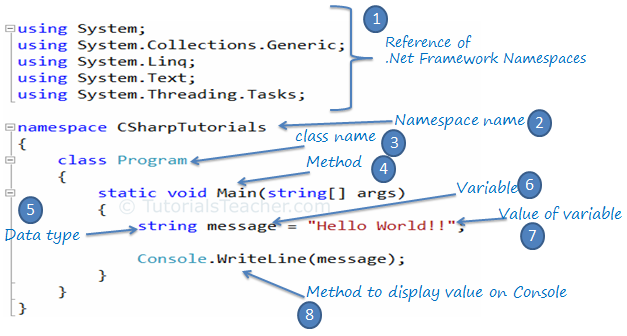
**Advantages of Visual Studio**

* Easy to create, edit and navigate different files or applications.
* Visual studio includes an excellent debugger that allows you to debug code easily. Step through the application code to understand it line by line, or identify problems in your code.
* Intellisense support for .Net Framework classes or custom classes.
* Nuget support for installing thrid party API/plug-ins in an application.
* Supports integration with many other third-party productive utilities which enhense the development quality and speed.
* Easy to configure, build and publish .NET applications.
* Provides ALM (Application Life-cycle Management) support for different phases of the development.

## Access Modifier Keywords:

Access modifiers are applied on the declaration of the class, method, properties, fields and other members. They define the accessibility of the class and its members.

| **Access Modifiers** | **Usage** |
| --- | --- |
| public | The Public modifier allows any part of the program in the same assembly or another assembly to access the type and its members. |
| private | The Private modifier restricts other parts of the program from accessing the type and its members. Only code in the same class or struct can access it. |
| internal | The Internal modifier allows other program code in the same assembly to access the type or its members. This is default access modifiers if no modifier is specified. |
| protected | The Protected modifier allows codes in the same class or a class that derives from that class to access the type or its members. |



# C# Class

A class is like a blueprint of specific object. In the real world, every object has some color, shape and functionalities.

public class MyClass

{

public string myField = string.Empty;

public MyClass()

{

}

public void MyMethod(int parameter1, string parameter2)

{

Console.WriteLine("First Parameter {0}, second parameter {1}",

parameter1, parameter2);

}

public int MyAutoImplementedProperty { get; set; }

private int myPropertyVar;

public int MyProperty

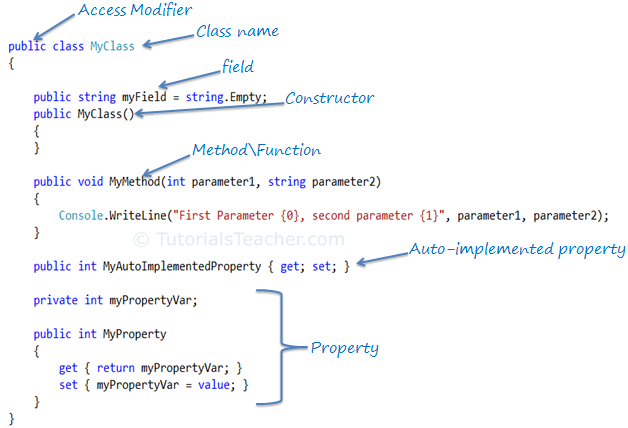
{

get { return myPropertyVar; }

set { myPropertyVar = value; }

}

}



**Lambda Expressions in C#**

Lambda expressions in C# are used like anonymous functions, with the difference that in Lambda expressions you don’t need to specify the type of the value that you input thus making it more flexible to use.

The ‘=>’ is the lambda operator which is used in all lambda expressions. The Lambda expression is divided into two parts, the left side is the input and the right is the expression.

**The Lambda Expressions can be of two types:**

**Expression Lambda:** Consists of the input and the expression.

*Syntax:*input => expression;

**Statement Lambda:** Consists of the input and a set of statements to be executed.

*Syntax:*input => { statements };

**Example 1:** In the code given below, we have a list of integer numbers. The first lambda expression evaluates every element’s square { x => x\*x } and the second is used to find which values are divisible by 3 { x => (x % 3) == 0 }. And the foreach loops are used for displaying.

using System;

using System.Collections.Generic;

using System.Linq;//Language Integrated Query

namespace Lambda\_Exressions

{class Program

{static void Main(string[] args)

{ // List to store numbers

List<int> numbers = new List<int>() {36, 71, 12, 15, 29, 18, 27, 17, 9, 34};

// foreach loop to display the list

Console.Write("The list : ");

foreach (var value in numbers)

{

Console.Write("{0} ", value);

}

Console.WriteLine();

// Using lambda expression

// to calculate square of

// each value in the list

var square = numbers.Select(x => x \* x);

// foreach loop to display squares

Console.Write("Squares : ");

foreach (var value in square)

{

Console.Write("{0} ", value);

}

Console.WriteLine();

// Using Lambda expression to

// find all numbers in the list

// divisible by 3

List<int> divBy3 = numbers.FindAll(x => (x % 3) == 0);

// foreach loop to display divBy3

Console.Write("Numbers Divisible by 3 : ");

foreach (var value in divBy3)

{

Console.Write("{0} ", value);

}

Console.WriteLine();

}

}

}

**Example 2:** Lambda expressions can also be used with user-defined classes. The code given below shows how to sort through a list based on an attribute of the class that the list is defined upon.

using System;

using System.Collections.Generic;

using System.Linq;

namespace Lambda\_Expression1

{ // User defined class Student

class Student

{ // properties rollNo and name

public int rollNo

{

get;

set;

}

public string name

{

get;

set;

}

}

class GFG

{ // Main Method

static void Main(string[] args)

{ // List with each element of type Student

List<Student> details = new List<Student>() {

new Student{ rollNo = 1, name = "Amma" },

new Student{ rollNo = 2, name = "Naghabhushan"},

new Student{ rollNo = 3, name = "Chaithu" },

new Student{ rollNo = 4, name = "RaghavendraBN" },

new Student{ rollNo = 5, name = "Bhanu" }

};

// To sort the details list

// based on name of student

// in acsending order

var newDetails = details.OrderBy(x => x.name);

foreach (var value in newDetails)

{

Console.WriteLine(value.rollNo + " " + value.name);

}

}

}

}

**C# collection** types are designed to store, manage and manipulate similar data more efficiently. Data manipulation includes adding, removing, finding, and inserting data in the collection. Collection types implement the following common functionality:

* Adding and inserting items to a collection
* Removing items from a collection
* Finding, sorting, searching items
* Replacing items
* Copy and clone collections and items
* Capacity and Count properties to find the capacity of the collection and number of items in the collection

.NET supports two types of collections, generic collections and non-generic collections.

The following table lists and matches these classes.

**Non-generic                          Generic**

 ArrayList     ------------->          List

 HashTable  ------------->          Dictionary

 SortedList   ------------->          SortedList

 Stack           ------------->          Stack

 Queue         ------------->          Queue

1. Non-Generic

In non-generic collections, each element can represent a value of a different type. The collection size is not fixed. Items from the collection can be added or removed at runtime.

C# ArrayList

ArrayList class is a collection that can be used for any types or objects. 

1. Arraylist is a class that is similar to an array, but it can be used to store values of various types.
2. An Arraylist doesn't have a specific size.
3. Any number of elements can be stored.

## 1.How to add an item to a C# ArrayList

The Add method on ArrayList appends a new item/element object to the end of the ArrayList. You can add elements in the ArrayList until memory runs out. The objects are stored in the managed heap. Let’s see an example of creating an ArrayList and add two elements using the Add() method of ArrayList.

using System;

using System.Collections;

namespace ArrayListApplication

{

class Program

{

static void Main(string[] args)

{

ArrayList resultList = new ArrayList();

resultList.Add(90);

resultList.Add(95);

Console.Read();

}

}

}

## 2. How to read elements from an ArrayList

You can read items from the ArrayList using a foreach loop

ArrayList items are stored on an index basis. Let’s see Figure 1.1 where the first item stored is at index 0 whereas the next item stored at the next index, in other words, index 1 so you can also retrieve items from an ArrayList using an index. Let’s see the following code.

using System;

using System.Collections;

namespace ReadElementsInCSharp

{

class Program

{

static void Main(string[] args)

{

ArrayList personList = new ArrayList();

personList.Add("Raghavendra");

personList.Add("Chaithanya");

foreach (var item in personList)

{

string arrayItem = string.Format($"Person's Name is {item}");

Console.WriteLine(arrayItem);

}

Console.Read();

}

}

}

2.1 ArrayList items are stored on an index basis.

using System;

using System.Collections;

namespace ReadElentInindex

{

class Program

{

static void Main(string[] args)

{

ArrayList personList = new ArrayList();

personList.Add("Sandeep");

personList.Add("Raviendra");

for (int i = 0; i < personList.Count; i++)

{

string arrayItem = string.Format($"Name is {personList[i]}");

Console.WriteLine(arrayItem);

}

Console.Read();

}

}

}

How to remove an item from an ArrayList

You can remove an item or items from an ArrayList. The ArrayList class provides four methods to remove an item or items from it. These methods are:

1. Remove
2. RemoveAt
3. RemoveRange
4. Clear

Let’s describe each one by one.

1. **Remove**

The Remove() method has one parameter that is the object type. You know that each item of the ArrayList is an object type so you need to pass that item in the Remove method to remove that item from the ArrayList.

using System;

using System.Collections;

namespace RemoveItemFromAryLst

{

class Program

{

static void Main(string[] args)

{

ArrayList personList = new ArrayList();

personList.Add("Raghavendra");

personList.Add("Chaithanya");

personList.Add("Sasken");

Console.WriteLine("=====Original List======");

for (int i = 0; i < personList.Count; i++)

{

string arrayItem = string.Format($"Name is {personList[i]}");

Console.WriteLine(arrayItem);

}

//remove first item from person list using index

personList.Remove("Sasken");

Console.WriteLine("=====Modified List======");

for (int i = 0; i < personList.Count; i++)

{

string arrayItem = string.Format($"Name is {personList[i]}");

Console.WriteLine(arrayItem);

}

Console.Read();

}

}

}

**2. RemoveAt**

In the previous method you saw that you can remove an item from the ArrayList using the Remove method. Notice one thing, that you need to remember the item to be removed. Suppose you have an ArrayList of complex objects that time you can’t remember the state of an object that will be removed. In this scenario, you can use the RemoveAt method of which you only need to pass the index number of the item.

using System;

using System.Collections;

namespace RemoveAtCsharp

{

class Program

{

static void Main(string[] args)

{

ArrayList personList = new ArrayList();

personList.Add("Sandeep");

personList.Add("Raviendra");

personList.Add("Shaijal");

Console.WriteLine("=====Original List======");

for (int i = 0; i < personList.Count; i++)

{

string arrayItem = string.Format($"Name is {personList[i]}");

Console.WriteLine(arrayItem);

}

//remove first item from person list using index

personList.RemoveAt(0);

Console.WriteLine("=====Modified List======");

for (int i = 0; i < personList.Count; i++)

{

string arrayItem = string.Format($"Name is {personList[i]}");

Console.WriteLine(arrayItem);

}

Console.Read();

}

}

}

**3. RemoveRange**

Each of the previous two methods removed a single item at a time. Suppose you want to remove more than one item from the ArrayList, then you need to call these methods multiple times. Alternatively you can use the RemoveRange method that can remove multiple items in a single call. The RemoveRange method has two arguments. The first argument is the beginning index number to be removed and the second parameter is the count yo be removed, in other words how many items you want to remove. But it removes items in sequence.

using System;

using System.Collections;

namespace RemoveRangeCsharp

{

class Program

{

static void Main(string[] args)

{

ArrayList personList = new ArrayList();

personList.Add("Sandeep");

personList.Add("Raviendra");

personList.Add("Shaijal");

Console.WriteLine("=====Original List======");

for (int i = 0; i < personList.Count; i++)

{

string arrayItem = string.Format($"Name is {personList[i]}");

Console.WriteLine(arrayItem);

}

//remove first two item from person list using index

personList.RemoveRange(0, 2);

Console.WriteLine("=====Modified List======");

for (int i = 0; i < personList.Count; i++)

{

string arrayItem = string.Format($"Name is {personList[i]}");

Console.WriteLine(arrayItem);

}

Console.Read();

}

}

}

**4. Clear**

The Clear method of ArrayList removes all the items from the ArrayList but doesn’t reduce the capacity of the ArrayList.

using System;

using System.Collections;

namespace ClearCsharp

{

class Program

{

static void Main(string[] args)

{

ArrayList personList = new ArrayList();

personList.Add("Sandeep");

personList.Add("Raviendra");

personList.Add("Shaijal");

int totalItems = personList.Count;

Console.WriteLine(string.Format($"Total Number Of Items in ArrayList: {totalItems}"));

//Remove all items from person list

personList.Clear();

totalItems = personList.Count;

Console.WriteLine(string.Format($"Total Number Of Items in ArrayList: {totalItems}"));

Console.Read();

}

}

}

# **Exception Handling in C#**

The exceptions are anomalies that occur during the execution of a program. They can be because of user, logic or system errors. If a user (programmer) does not provide a mechanism to handle these anomalies, the .NET runtime environment provides a default mechanism, which terminates the program execution.

Exception handling in C# and .NET framework, suppoted by the try catch and finaly block is a mechanism to detect and handle run-time errors in code. The .NET framework provides built-in classes for common exceptions.

try..catch..finally

1. **try**
2. {
3. // Statement which can cause an exception.
4. }
5. **catch**(Type x)
6. {
7. // Statements for handling the exception
8. }
9. **finally**
10. {
11. //Any cleanup code
12. }

DivideByZeroException

using System;

namespace ExceptionHandling

{

class Program

{

public static void Main()

{

int x = 0;

int div = 0;

try

{

div = 100 / x;

Console.WriteLine("This line will not executed");

}

catch (DivideByZeroException)

{

Console.WriteLine("Exception occured");

}

Console.WriteLine($"Result is {div}");

}

}

}

Multiple Catch Blocks

A try block can throw multiple exceptions, which can handle by using multiple catch blocks. Remember that more specialized catch block should come before a generalized one. Otherwise the compiler will show a compilation error.

using System;

namespace MultipleBlock

{

class Program

{

public static void Main()

{

int x = 0;

int div = 0;

try

{

div = 100 / x;

Console.WriteLine("Not executed line");

}

catch (DivideByZeroException de)

{

Console.WriteLine("DivideByZeroException");

}

catch (Exception)

{

Console.WriteLine("Exception");

}

finally

{

Console.WriteLine("Finally Block");

}

Console.WriteLine($"Result is {div}");

}

}

}

User-defined Exceptions

In C#, it is possible to create our own exception class. But Exception must be the ultimate base class for all exceptions in C#. So the user-defined exception classes must inherit from either Exception class or one of its standard derived classes.

using System;

class MyException : Exception

{

public MyException(string str)

{

Console.WriteLine("User defined exception");

}

}

class MyClient

{

public static void Main()

{

try

{

throw new MyException("Raghavendra");

}

catch (Exception e)

{

Console.WriteLine("Exception caught here" + e.ToString());

}

Console.WriteLine("LAST STATEMENT");

}

}

## C# data type

A data type is a set of values, and the allowable operations on those values.

The two fundamental data types in C# are value types and reference types. Primitive types (except strings), enumerations, and structures are value types. Classes, strings, interfaces, arrays, and delegates are reference types. Every type has a default value. Reference types are created on the Heap. The lifetime of the reference type is managed by the .NET framework. The default value for reference types is null reference. Assignment to a variable of a reference type creates a copy of the reference rather than a copy of the referenced value. Value types are created on the stack. The lifetime is determined by the lifetime of the variable. Assignment to a variable of a value type creates a copy of the value being assigned. Value types have different default values. For example, boolean default value is false, decimal 0, string an empty string "".

## C# Boolean values

There is a duality built in our world. There is a Heaven and Earth, water and fire, Yin and Yang, man and woman, love and hatred. In C# the bool data type is a primitive data type having one of two values: true or false. This is a fundamental data type that is very common in computer programs.

Happy parents are waiting a child to be born. They have chosen a name for both possibilities. If it is going to be a boy, they have chosen John. If it is going to be a girl, they have chosen Victoria.

Program.cs

using System;

namespace BooleanType

{

class Program

{

static void Main(string[] args)

{

var random = new Random();

var male = Convert.ToBoolean(random.Next(0, 2));

if (male)

{

Console.WriteLine("We will use name John");

}

else

{

Console.WriteLine("We will use name Victoria");

}

}

}

}

The program uses a random number generator to simulate our case.

var random = new Random();

We create a Random object which is used to compute random numbers. It is part of the System namespace.

var male = Convert.ToBoolean(random.Next(0, 2));

The Next() method returns a random number within a specified range. The lower bound is included, the upper bound is not. In other words, we receive either 0 or 1. Later the Convert() method converts these values to boolean ones, 0 to false and 1 to true.

if (male)

{

Console.WriteLine("We will use name John");

} else

{

Console.WriteLine("We will use name Victoria");

}

If the male variable is set to true, we choose the name John. Otherwise, we choose the name Victoria. Control structures like if/else statements work with boolean values.

$ dotnet run

We will use name John

$ dotnet run

We will use name John

$ dotnet run

We will use name Victoria

Running the program several times gives this sample output.

## C# integers

Integers are a subset of the real numbers. They are written without a fraction or a decimal component. Integers fall within a set Z = {..., -2, -1, 0, 1, 2, ...}. Integers are infinite.

In computer languages, integers are primitive data types. Computers can practically work only with a subset of integer values, because computers have finite capacity. Integers are used to count discrete entities. We can have 3, 4, 6 humans, but we cannot have 3.33 humans. We can have 3.33 kilograms.

|  |  |  |  |
| --- | --- | --- | --- |
| **VB Alias** | **.NET Type** | **Size** | **Range** |
| sbyte | System.SByte | 1 byte | -128 to 127 |
| byte | System.Byte | 1 byte | 0 to 255 |
| short | System.Int16 | 2 bytes | -32,768 to 32,767 |
| ushort | System.UInt16 | 2 bytes | 0 to 65,535 |
| int | System.Int32 | 4 bytes | -2,147,483,648 to 2,147,483,647 |
| uint | System.UInt32 | 4 bytes | 0 to 4,294,967,295 |
| long | System.Int64 | 8 bytes | -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |
| ulong | System.UInt64 | 8 bytes | 0 to 18,446,744,073,709,551,615 |

These integer types may be used according to our needs. No one, (except perhaps for some biblical people), can be older than 120, 130 years. We can then use the byte type for age variable in a program. This will save some memory.

### Discrete entities

If we work with integers, we deal with discrete entities. We would use integers to count apples.

Program.cs

using System;

namespace Apples

{

class Program

{

static void Main(string[] args)

{

int baskets = 16;

int applesInBasket = 24;

int total = baskets \* applesInBasket;

Console.WriteLine("There are total of {0} apples", total);

}

}

}

In our program, we count the total amount of apples. We use the multiplication operation.

int baskets = 16;

int applesInBasket = 24;

The number of baskets and the number of apples in each basket are integer values.

int total = baskets \* applesInBasket;

Multiplying those values we get an integer too.

$ dotnet run

There are total of 384 apples

This is the output of the program.

### C# integer notations

Integers can be specified in three different notations in C#: decimal, hexadecimal, and binary. There are no notations for octal values. Decimal numbers are used normally as we know them. Hexadecimal numbers are preceded with 0x characters, binary with 0b.

Program.cs

using System;

namespace IntegerNotations

{

class Program

{

static void Main(string[] args)

{

int num1 = 31;

int num2 = 0x31;

int num3 = 0b1101;

Console.WriteLine(num1);

Console.WriteLine(num2);

Console.WriteLine(num3);

}

}

}

In the program, we have three integers expressed in three different notations.

$ dotnet run

31

49

13

The default notation is the decimal. The program shows these two numbers in decimal. In other words, hexadecimal 0x31 is 49 decimal.

### Using underscores

C# allows to use underscore characters for numeric literals to increase readability of the values.

Program.cs

using System;

namespace UnderscoreLiterals

{

class Program

{

static void Main(string[] args)

{

var num = 234\_321\_000;

Console.WriteLine(num);

}

}

}

The program uses an integer literal with underscore character to improve the readability of the value.

### Arithmetic overflow

An arithmetic overflow is a condition that occurs when a calculation produces a result that is greater in magnitude than that which a given register or storage location can store or represent.

Program.cs

using System;

namespace OverFlow

{

class Program

{

static void Main(string[] args)

{

byte a = 254;

Console.WriteLine(a);

a++;

Console.WriteLine(a);

a++;

Console.WriteLine(a);

a++;

Console.WriteLine(a);

}

}

}

In this example, we try to assign a value beyond the range of a data type. This leads to an arithmetic overflow.

$ dotnet run

254

255

0

1

When an overflow occurs, the variable is reset to the lower bound of the data type. (In case of a byte type it is zero.)

With the checked keyword, we can enforce an exception when the overflow occurs.

Program.cs

using System;

namespace OverflowChecked

{

class Program

{

static void Main(string[] args)

{

checked {

byte a = 254;

Console.WriteLine(a);

a++;

Console.WriteLine(a);

a++;

Console.WriteLine(a);

a++;

Console.WriteLine(a);

}

}

}

}

In the example, the statements are placed in the body of the checked block.

$ dotnet run

254

255

Unhandled Exception: System.OverflowException: Arithmetic operation resulted in an overflow.

...

This time a System.OverflowException is thrown.

## C# floating point numbers

Floating point numbers represent real numbers in computing. Real numbers measure continuous quantities, like weight, height, or speed. In C# we have three floating point types: float, double, and decimal.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **C# Alias** | **.NET Type** | **Size** | **Precision** | **Range** |
| float | System.Single | 4 bytes | 7 digits | +-1.5 x 10-45 to +-3.4 x 1038 |
| double | System.Double | 8 bytes | 15-16 digits | +-5.0 x 10-324 to +-1.7 x 10308 |
| decimal | System.Decimal | 16 bytes | 28-29 decimal places | +-1.0 x 10-28to +-7.9 x 1028 |

The above table gives the characteristics of the floating point types.

By default, real numbers are double in C# programs. To use a different type, we must use a suffix. The F/f for float numbers and M/m for decimal numbers.

Program.cs

using System;

namespace Floats

{

class Program

{

static void Main(string[] args)

{

float n1 = 1.234f;

double n2 = 1.234;

decimal n3 = 1.234m;

Console.WriteLine(n1);

Console.WriteLine(n2);

Console.WriteLine(n3);

Console.WriteLine(n1.GetType());

Console.WriteLine(n2.GetType());

Console.WriteLine(n3.GetType());

}

}

}

In the above program, we use three different literal notations for floating point numbers.

float n1 = 1.234f;

The f suffix is used for a float number.

double n2 = 1.234;

If we do not use a suffix, then it is a double number. We can optionally use the d suffix.

Console.WriteLine(n1.GetType());

The GetType() method returns the type of the number.

$ dotnet run

1.234

1.234

1.234

System.Single

System.Double

System.Decimal

This is the output.

We can use various syntax to create floating point values.

Program.cs

using System;

namespace Notations

{

class Program

{

static void Main(string[] args)

{

float n1 = 1.234f;

float n2 = 1.2e-3f;

float n3 = (float)1 / 3;

Console.WriteLine(n1);

Console.WriteLine(n2);

Console.WriteLine(n3);

}

}

}

We have three ways to create floating point values. The first is the 'normal' way using a decimal point. The second uses a scientific notation. And the last one as a result of a numerical operation.

float n2 = 1.2e-3f;

This is the scientific notation for floating point numbers. Also known as exponential notation, it is a way of writing numbers too large or small to be conveniently written in standard decimal notation.

float n3 = (float) 1 / 3;

The (float) construct is called casting. The division operation returns integer numbers by default. By casting we get a float number.

$ dotnet run

1.234

0.0012

0.3333333

This is the output of the above program.

The float and double types are inexact.

Program.cs

using System;

namespace InExact

{

class Program

{

static void Main(string[] args)

{

double n1 = 0.1 + 0.1 + 0.1;

double n2 = 1 / 3.0;

if (n1 == n2)

{

Console.WriteLine("Numbers are equal");

}

else

{

Console.WriteLine("Numbers are not equal");

}

}

}

}

Caution should be exercised when comparing floating point values.

$ dotnet run

Numbers are not equal

And the numbers are not equal.

Let's say a sprinter for 100m ran 9.87s. What is his speed in km/h?

Program.cs

using System;

namespace Sprinter

{

class Program

{

static void Main(string[] args)

{

float distance = 0.1f;

float time = 9.87f / 3600;

float speed = distance / time;

Console.WriteLine("The average speed of a sprinter is {0} km/h", speed);

}

}

}

In this example, it is necessary to use floating point values.

float distance = 0.1f;

100 m is 0.1 km.

float time = 9.87f / 3600;

9.87 s is 9.87/(60\*60) h.

float speed = distance / time;

To get the speed, we divide the distance by the time.

$ dotnet run

The average speed of a sprinter is 36.47416 km/h

This is the output of the program.

## C# enumerations

Enumerated type (also called enumeration or enum) is a data type consisting of a set of named values. A variable that has been declared as having an enumerated type can be assigned any of the enumerators as a value. Enumerations make the code more readable.

Program.cs

using System;

namespace Enumerations

{

enum Days

{

Monday,

Tuesday,

Wednesday,

Thursday,

Friday,

Saturday,

Sunday

}

class Program

{

static void Main(string[] args)

{

Days day = Days.Monday;

if (day == Days.Monday)

{

Console.WriteLine("It is Monday");

}

Console.WriteLine(day);

foreach (int i in Enum.GetValues(typeof(Days)))

{

Console.WriteLine(i);

}

}

}

}

In our code example, we create an enumeration for week days.

enum Days

{

Monday,

Tuesday,

Wednesday,

Thursday,

Friday,

Saturday,

Sunday

}

The enumeration is created with a enum keyword. The Monday, Tuesday, ... barewords store in fact numbers 0..6.

Days day = Days.Monday;

We have a variable called day which is of the enumerated type Days. It is initialized to Monday.

if (day == Days.Monday)

{

Console.WriteLine("It is Monday");

}

This code is more readable than comparing a day variable to some number.

Console.WriteLine(day);

This line prints Monday to the console.

foreach (int i in Enum.GetValues(typeof(Days)))

{

Console.WriteLine(i);

}

This loop prints 0..6 to the console. We get underlying types of the enum values. For a computer, an enum is just a number. The typeof is an operator used to obtain the System.Type object for a type. It is needed by the GetValues() method. This method returns an array of the values of a specified enumeration. And the foreach keyword goes through the array, element by element and prints them to the terminal.

We further work with enumerations.

Program.cs

using System;

using System;

namespace Seasons

{

public enum Seasons : byte

{

Spring = 1,

Summer = 2,

Autumn = 3,

Winter = 4

}

class Program

{

static void Main(string[] args)

{

Seasons s1 = Seasons.Spring;

Seasons s2 = Seasons.Autumn;

Console.WriteLine(s1);

Console.WriteLine(s2);

}

}

}

Seasons can be easily used as enums. We can specify the underlying type for the enum and we can give exact values for them.

public enum Seasons : byte

{

Spring = 1,

Summer = 2,

Autumn = 3,

Winter = 4

}

With a colon and a data type we specify the underlying type for the enum. We also give each member a specific number.

Console.WriteLine(s1);

Console.WriteLine(s2);

These two lines print the enum values to the console.

$ dotnet run

Spring

Autumn

This is the output of the program.

## C# strings and chars

The string is a data type representing textual data in computer programs. A string in C# is a sequence of Unicode characters. A char is a single Unicode character. Strings are enclosed by double quotes.

Since strings are very important in every programming language, we will dedicate a whole chapter to them. Here we only present a small example.

Program.cs

using System;

namespace Strings

{

class Program

{

static void Main(string[] args)

{

string word = "ZetCode";

char c = word[0];

Console.WriteLine(c);

}

}

}

The program prints 'Z' character to the terminal.

string word = "ZetCode";

Here we create a string variable and assign it the "ZetCode" value.

char c = word[0];

A string is an array of Unicode characters. We can use the array access notation to get a specific character from the string. The number inside the square brackets is the index into the array of characters. The index is counted from zero. It means that the first character has index 0.

$ dotnet run

Z

The program prints the first character of the "ZetCode" string to the console.

## C# arrays

The array is a complex data type which handles a collection of elements. Each of the elements can be accessed by an index. All the elements of an array must be of the same data type.

We dedicate a whole chapter to arrays; here we show only a small example.

Program.cs

using System;

namespace ArrayEx

{

class Program

{

static void Main(string[] args)

{

int[] numbers = new int[5];

numbers[0] = 3;

numbers[1] = 2;

numbers[2] = 1;

numbers[3] = 5;

numbers[4] = 6;

int len = numbers.Length;

for (int i = 0; i < len; i++)

{

Console.WriteLine(numbers[i]);

}

}

}

}

In this example, we declare an array, fill it with data and then print the contents of the array to the console.

int[] numbers = new int[5];

We declare an integer array which can store up to five integers. So we have an array of five elements, with indexes 0..4.

numbers[0] = 3;

numbers[1] = 2;

numbers[2] = 1;

numbers[3] = 5;

numbers[4] = 6;

Here we assign values to the created array. We can access the elements of an array by the array access notation. It consists of the array name followed by square brackets. Inside the brackets we specify the index to the element that we want.

int len = numbers.Length;

Each array has a Length property which returns the number of elements in the array.

for (int i=0; i<len; i++)

{

Console.WriteLine(numbers[i]);

}

We traverse the array and print the data to the console.

## C# DateTime

The DateTime is a value type. It represents an instant in time, typically expressed as a date and time of day.

Program.cs

using System;

namespace DateTimeEx

{

class Program

{

static void Main(string[] args)

{

DateTime now = DateTime.Now;

System.Console.WriteLine(now);

System.Console.WriteLine(now.ToShortDateString());

System.Console.WriteLine(now.ToShortTimeString());

}

}

}

We show today's date in three different formats: date & time, date, and time.

DateTime now = DateTime.Now;

Gets a DateTime object that is set to the current date and time on this computer, expressed as the local time.

System.Console.WriteLine(now);

This line prints the date in full format.

System.Console.WriteLine(now.ToShortDateString());

System.Console.WriteLine(now.ToShortTimeString());

The ToShortDateString() returns a short date string format, the ToShortTimeString() returns a short time string format.

$ dotnet run

12/5/2018 8:09:56 PM

12/5/2018

8:09 PM

We see the output of the example.

## C# type casting

We often work with multiple data types at once. Converting one data type to another one is a common job in programming. Type conversion or typecasting refers to changing an entity of one data type into another. There are two types of conversion: implicit and explicit. Implicit type conversion, also known as coercion, is an automatic type conversion by the compiler.

Program.cs

using System;

namespace ImplicitTypeConversion

{

class Program

{

static void Main(string[] args)

{

int val1 = 0;

byte val2 = 15;

val1 = val2;

Console.WriteLine(val1.GetType());

Console.WriteLine(val2.GetType());

Console.WriteLine(12 + 12.5);

Console.WriteLine("12" + 12);

}

}

}

In this example, we have several implicit conversions.

val1 = val2;

Here we work with two different types: int and byte. We assign a byte value to an int value. It is a widening operation. The int values have four bytes; byte values have only one byte. Widening conversions are allowed. If we wanted to assign a int to a byte, this would be a shortening conversion. Implicit shortening conversions are not allowed by C# compiler. This is because in implicit shortening conversion we could unintentionally loose precision. We can do shortening conversions, but we must inform the compiler about it. That we know what we are doing. It can be done with explicit conversion.

Console.WriteLine(12 + 12.5);

We add two values: one integer and one floating point value. The result is a floating point value. It is a widening implicit conversion.

Console.WriteLine("12" + 12);

The result is 1212. An integer is converted to a string and the two strings are concatenated.

Next we will show some explicit conversions in C#.

Program.cs

using System;

namespace ExplicitTypeConversion

{

class Program

{

static void Main(string[] args)

{

double b = 13.5;

float a = (float) b;

float c = (int) a;

Console.WriteLine(a);

Console.WriteLine(b);

Console.WriteLine(c);

}

}

}

We have three values. We do some explicit conversions with these values.

float a = (float) b;

We convert a double value to a float value. Explicit conversion is done by specifying the intended type between two round brackets. In this case, no precision is lost. Number 13.5 can be safely assigned to both types.

float c = (int) a;

We convert a float value to int value. In this statement, we loose some precision: 13.5 becomes 13.

$ dotnet run

13.5

13.5

13

We see the output of the program.

## C# Nullable types

Value types cannot be assigned a null literal, reference types can. Applications that work with databases deal with the null value. Because of this, special nullable types were introduced into the C# language. Nullable types are instances of the System.Nullable<T> struct.

Program.cs

using System;

class NullableType

{

static void Main()

{

Nullable<bool> male = null;

int? age = null;

Console.WriteLine(male.HasValue);

Console.WriteLine(age.HasValue);

}

}

A simple example demonstrating nullable types.

Nullable<bool> male = null;

int? age = null;

There are two ways how to declare a nullable type. Either with the Nullable<T> generic structure in which the type is specified between the angle brackets, or we can use a question mark after the type. The latter is in fact a shorthand for the first notation.

$ dotnet run

False

False

This is the output of the example.

## C# Convert & Parse methods

There are two groups of methods which are used to convert values.

Program.cs

using System;

namespace ConvertEx

{

class Program

{

static void Main(string[] args)

{

Console.WriteLine(Convert.ToBoolean(0.3));

Console.WriteLine(Convert.ToBoolean(3));

Console.WriteLine(Convert.ToBoolean(0));

Console.WriteLine(Convert.ToBoolean(-1));

Console.WriteLine(Convert.ToInt32("452"));

Console.WriteLine(Convert.ToInt32(34.5));

}

}

}

The Convert class has many methods for converting values. We use two of them.

Console.WriteLine(Convert.ToBoolean(0.3));

We convert a double value to a bool value.

Console.WriteLine(Convert.ToInt32("452"));

And here we convert a string to an int.

$ dotnet run

True

True

False

True

452

34

This is the output.

Program.cs

using System;

namespace ParseEx

{

class Program

{

static void Main(string[] args)

{

Console.WriteLine(Int32.Parse("34"));

Console.WriteLine(Int32.Parse("-34"));

Console.WriteLine(Int32.Parse("+34"));

}

}

}

Converting strings to integers is a very common task. We often do such conversions when we fetch values from databases or GUI components.

Console.WriteLine(Int32.Parse("34"));

We use the Parse() method of the Int32 class to convert a string to int value.

$ dotnet run

34

-34

34

This is the output.